Programación Concurrente

Teoría 9

Facultad de Informática
UNLP

Links al archivo con audio

La teoría con los audios está en formato MP4. Debe descargar los archivos comprimidos de los siguientes links:

- Librería Pthreads:
https://drive.google.com/uc?id=1T2TBPMgrc1ax0z_vrDWMiSaSXJOTJ9GR&export=download

- Semáforos y monitores en Pthreads:
https://drive.google.com/uc?id=1gLhJodkLt8ukBGLPNgauoHRX2HGX0sN9&export=download

- Librería para Pasaje de Mensajes (MPI):
https://drive.google.com/uc?id=1tlOM5BaPD2KSlRIUVYWnwO-8SDqLPlE8&export=download

Librería para Memoria Compartida

Pthreads

Pthreads

Thread: proceso "liviano" que tiene su propio contador de programa y su pila de ejecución, pero no controla el "contexto pesado" (por ejemplo, las tablas de página).

- Algunos sistemas operativos y lenguajes proveen mecanismos para permitir la programación de aplicaciones "multithreading".
- En principio estos mecanismos fueron heterogéneos y poco portables $\Rightarrow$ a mediados de los 90 la organización POSIX auspició el desarrolló de una biblioteca en C para multithreading **(*Pthreads*).**
- Pthreads es una bilbioteca para programación paralela en ***memoria compartida,*** se pueden crear threads, asignarles atributos, darlos por terminados, identificarlos, etc.

POSIX – API de Threads

- Numerosas APIs para el manejo de Threads.
- Normalmente llamada Pthreads, POSIX a emergido como un API estandard para manejo de Threads, provista por la mayoría de los vendedores.
- Los conceptos que se discutirán son independientes de la API y pueden ser igualmente válidos para utilizar JAVA Threads, NT Threads, Solaris Threads, etc.
- Funciones reentrantes.

Pthreads - Creación y terminación

- Pthreads provee funciones básicas para especificar concurrencia:

```
#include <pthread.h>

int pthread_create (pthread_t  *thread_handle, const pthread_attr_t  *attribute,
                    void * (*thread_function)(void *), void *arg);

int pthread_exit (void *res);

int pthread_join (pthread_t   thread, void **ptr);

int pthread_cancel (pthread_t thread);
```

- El “main” debe esperar a que todos los threads terminen.

Pthreads – Primitivas de Sincronización
Exclusión mutua

- Las secciones críticas se implementan en Pthreads utilizando ***mutex locks*** (bloqueo por exclusión mutua) por medio de variables ***mutex***.
- Una variable ***mutex*** tienen dos estados: locked (bloqueado) and unlocked (desbloqueado). En cualquier instante, sólo UN thread puede bloquear un *mutex*. *Lock* es una operación atómica.
- Para entrar en la sección crítica un Thread debe lograr tener control del *mutex* (bloquearlo).
- Cuando un Thread sale de la SC debe desbloquear el *mutex*.
- Todos los *mutex* deben inicializarse como desbloqueados.

Pthreads – Primitivas de Sincronización
Exclusión mutua

- La API Pthreads provee las siguientes funciones para manejar los *mutex*:

```
int pthread_mutex_lock ( pthread_mutex_t *mutex);

int pthread_mutex_unlock (pthread_mutex_t *mutex);

int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *lock_attr);
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Exclusión Mutua

El escenario de productores-consumidores impone las siguientes restricciones:

- Un thread productor no debe sobrescribir el buffer compartido cuando el elemento anterior no ha sido tomado por un thread consumidor.
- Un thread consumidor no puede tomar nada de la estructura compartida hasta no estar seguro de que se ha producido algo anteriormente.
- Los consumidores deben excluirse entre sí.
- Los productores deben excluirse entre sí.
- En este ejemplo el buffer es de tamaño 1.

Pthreads - Primitivas de Sincronización Productores/Consumidores con Exclusión Mutua

Main de la solución al problema de productores-consumidores.

```
pthread_mutex_t    mutex;
int hayElemento;
tipo_elemento Buffer;
...
main()
   { hayElemento= 0;
     pthread_init ();
     pthread_mutex_init(&mutex, NULL);

       /* Create y join de threads productores y consumidores*/
    }
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Exclusión Mutua

Código para los *productores*.

```
void *productor (void *datos)
  {  tipo_elemento elem;
     int ok;
     ....
     while (true)
           { ok = 0;
             generar_elemento(&elem);
             while (ok = = 0)
                {  pthread_mutex_lock(&mutex);
                   if (hayElemento== 0)
                      {  Buffer = elem;
                         hayElemento = 1;
                         ok = 1;
                       }
                   pthread_mutex_unlock(&mutex);
                 }
           }
  }
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Exclusión Mutua

Código para los *consumidores*.

```
void *consumidor(void *datos)
  {  int ok;
     tipo_elemento elem;
     ....
     while (true)
        { ok = 0;
          while (ok == 0)
              {  pthread_mutex_lock(&mutex);
                 if (hayElemento == 1)
                    { elem = Buffer;
                      hayElemento = 0;
                      ok = 1;
                     }
                 pthread_mutex_unlock(&mutex);
              }
           procesar_elemento(elem);
        }
   }
```

Pthreads - Primitivas de Sincronización
Tipos de Exclusión Mutua (*Mutex*)

- Pthreads soporta tres tipos de Mutexs (Locks): Normal, Recursive y Error Check
  - ✓ Un Mutex con el atributo Normal NO permite que un thread que lo tienen bloqueado vuelva a hacer un lock sobre él (deadlock).
  - ✓ Un Mutex con el atributo Recursive SI permite que un thread que lo tienen bloqueado vuelva a hacer un lock sobre él. Simplemente incrementa una cuenta de control.
  - ✓ Un Mutex con el atributo ErrorCheck responde con un reporte de error al intento de un segundo bloqueo por el mismo thread.
- El tipo de Mutex puede setearse entre los atributos antes de su inicialización.

Pthreads - Primitivas de Sincronización
Overhead de Bloqueos por Exclusión Mutua

- Los locks representan puntos de serialización → si dentro de las secciones críticas ponemos segmentos largos de programa tendremos una degradación importante de perfomance.

- A menudo se puede reducir el overhead por espera ociosa, utilizando la función pthread_mutex_trylock. Retorna el control informando si pudo hacer o no el lock.

  int pthread_mutex_trylock (pthread_mutex_t *mutex_lock).

  - ✓ Evita tiempos ociosos.
  - ✓ Menos costoso por no tener que manejar las colas de espera.

Pthreads - Primitivas de Sincronización
Variables Condición

- Podemos utilizar variables de condición para que un thread se autobloquee hasta que se alcance un estado determinado del programa.
- Cada variable de condición estará asociada con un predicado. Cuando el predicado se convierte en verdadero (TRUE) la variable de condición da una señal para el/los threads que están esperando por el cambio de estado de la condición.
- Una única variable de condición puede asociarse a varios predicados (difícil el debug).
- Una variable de condición siempre tiene un mutex asociada a ella. Cada thread bloquea este mutex y testea el predicado definido sobre la variable compartida.
- Si el predicado es falso, el thread espera en la variable condición utilizando la función pthread_cond_wait (NO USA CPU).

Pthreads - Primitivas de Sincronización
Variables Condición

La API Pthreads provee las siguientes funciones para manejar las variables condición:

```
int pthread_cond_wait ( pthread_cond_t *cond,
                        pthread_mutex_t *mutex)

int pthread_cond_timedwait ( pthread_cond_t *cond,
                             pthread_mutex_t *mutex
                             const struct timespec *abstime)

int pthread_cond_signal (pthread_cond_t *cond)

int pthread_cond_broadcast (pthread_cond_t *cond)

int pthread_cond_init ( pthread_cond_t *cond,
                        const pthread_condattr_t *attr)

int pthread_cond_destroy (pthread_cond_t *cond)
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Variables Condición

Main de la solución al problema de productores-consumidores.

```
pthread_cond_t vacio, lleno;
pthread_mutex_t mutex;
int hayElemento;
tipo_elemento Buffer;
...
main()
    { ...
      hayElemento= 0;
      pthread_init();
      pthread_cond_init(&vacio, NULL);
      pthread_cond_init(&lleno, NULL);
      pthread_mutex_init(&mutex, NULL);
      ...
    }
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Variables Condición

Código para los *productores*.

```
void *productor(void *datos)
   { tipo_element elem;

     while (true)
        { generar_elemento(elem);
          pthread_mutex_lock (&mutex);
          while (hayElemento = = 1)
                pthread_cond_wait (&vacio, &mutex);
           Buffer = elem;
           hayElemento = 1;
           pthread_cond_signal (&lleno);
           pthread_mutex_unlock (&mutex);
        }
   }
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Variables Condición

Código para los *consumidores*.

```
void *consumidor(void *datos)
   {  tipo_element elem;

      while (true))
       { pthread_mutex_lock (&mutex);
         while (hayElemento = = 0)
                  pthread_cond_wait (&lleno, &mutex);
         elem= Buffer;
         hayElemento = 0;
         pthread_cond_signal (&vacio);
         pthread_mutex_unlock (&mutex);
         procesar_elemento(elem);
       }
  }
```

Pthreads – Atributos y sincronización

- La API Pthreads permite que se pueda cambiar los atributos por defecto de las entidades, utilizando attributes objects.
- Un attribute object es una estructura de datos que describe las propiedades de la entidad en cuestión (thread, mutex, variable de condición).
- Una vez que estas propiedades están establecidas, el attribute object es pasado al método que inicializa la entidad.
- Ventajas
  - ✓ Esta posibilidad mejora la modularidad.
  - ✓ Facilidad de modificación del código.

Pthreads – Atributos para Threads

- La API Pthreads provee las siguientes funciones para manejar los atributos para Threads:

```
int pthread_attr_init (pthread_attr_t  *attr);

int pthread_attr_destroy (pthread_attr_t  *attr);
```

- Las propiedades asociadas con el *attribute object* pueden ser cambiadas con las siguientes funciones:

```
pthread_attr_setdetachstate

pthread_attr_setguardsize_np

pthread_attr_setstacksize

pthread_attr_setinheritsched

pthread_attr_setschedpolicy

pthread_attr_setschedparam
```

Pthreads – Atributos para *Mutex*

- La API Pthreads provee las siguientes funciones para manejar los atributos para Mutex:

```
int pthread_mutexattr_init (pthread_mutexattr_t  *attr);

int pthread_mutexattr_settype_np ( pthread_mutexattr_t *attr,   int type);
```

- Aquí *type* especifica el tipo de *mutex* y puede tomar los valores:
  - PTHREAD_MUTEX_NORMAL_NP
  - PTHREAD_MUTEX_RECURSIVE_NP
  - PTHREAD_MUTEX_ERRORCHECK_NP

Semáforos en Pthreads

Semáforos con Pthreads

- Los threads pueden sincronizar por semáforos (*librería semaphore.h*).
- Declaración y operaciones con semáforos en Pthreads:
  - ✓ sem_t semaforo→ se declaran globales a los threads.
  - ✓ sem_init (&semaforo, alcance, inicial) → en esta operación se inicializa el semáforo semaforo. Inicial es el valor con que se inicializa el semáforo. Alcance indica si es compartido por los hilos de un único proceso (0) o por los de todos los procesos (≠ 0).
  - ✓ sem_wait(&semaforo) → equivale al P.
  - ✓ sem_post(&semaforo) → equivale al V.
  - ✓ Existen funciones extras para: wait condicional, obtener el valor de un semáforo y destruir un semáforo (ESTE TIPO DE FUNCIONES EXTRAS NO SE PUEDEN USAR EN LA PRÁCTICA DE LA MATERIA).

Semáforos con Pthreads

Productor / consumidor

- Las funciones de ***Productor*** y ***Consumidor*** serán ejecutadas por threads independientes.
- Acceden a un buffer compartido (***datos***).
- El productor deposita una secuencia de enteros de ***1*** a ***numItems*** en el buffer.
- El consumidor busca estos valores y los suma.
- Los semáforos ***vacio*** y ***lleno*** garantizan el acceso alternativo de productor y consumidor sobre el buffer.

```
#include <pthread.h>
#include <semaphore.h>
#define SHARED 1

void *Productor(void *);
void *Consumidor(void *);

sem_t  vacio, lleno;
int dato, numItems;
```

```
int main(int argc, char * argv[ ])
{    ……
     sem_init (&vacio, SHARED, 1);
     sem_init (&lleno, SHARED, 0);
     ……
     pthread_create (&pid, &attr,  Productor, NULL);
     pthread_create (&cid, &attr, Consumidor, NULL);
     pthread_join (pid, NULL);
     pthread_join (cid, NULL);
}
```

Semáforos con Pthreads

Productor / consumidor

```
void *Productor (void *arg)
{ int item;
  for (item = 1; item <= numItems; item++)
     {   sem_wait(&vacio);
         dato = item;
         sem_post(&lleno);
      }
   pthreads_exit();
}

void *Consumidor (void *arg)
{   int total = 0, item, aux;
    for (item = 1; item <= numItems; item++)
       { sem_wait(&lleno);
         aux = dato;
         sem_post(&vacio);
         total = total + aux;
       }
    printf("TOTAL:  %d\n", total);
    pthreads_exit();
}
```

Monitores en Pthreads

Monitores con Pthreads

- Pthreads no permite manejar la Exclusión Mutua por medio de las variables *mutex*.
- Pthreads nos permite manejar la Sincronización por Condición utilizando *variables condición* para que un *thread* se auto bloquee hasta que se alcance un estado determinado del programa. Una variable de condición siempre tiene un *mutex* asociada a ella.
- Pthreads no posee "*Monitores*", pero con las dos herramientas que mencionamos se puede similar el uso de monitores: con mutex se hace la exclusión mutua que nos brindaba implicitamente el monitor, y con las variables condición la sincronización.
  - El acceso exclusive al monitor se simula usando una variable *mutex* la cual se bloquea antes del llamada al *procedure* y se desbloquea al terminar el mismo (una variable *mutex* diferente para cada monitor).
  - Cada llamado de un proceso a un procedure de un monitor debe ser reemplazado por el código de ese procedure.

Monitores con Pthreads

Ejemplo: *Lectores y escritores*

- Esta es la solución que vimos en la teoría de monitores para el problema de lectores/escritores. Ahora veremos como simularla en ***Pthreads***.

```
monitor Controlador
{  int nr = 0, nw = 0, dr = 0, dw = 0;
   cond  ok_leer, ok_escribir

   procedure pedido_leer( )
    { if (nw > 0)
                    { dr = dr +1;
                      wait (ok_leer);
                    }
      else  nr = nr + 1;
    }

   procedure libera_leer( )
    { nr = nr - 1;
      if (nr == 0 and dw > 0)
                { dw = dw -1;
                  signal (ok_escribir);
                  nw = nw +1;
                }
    }

   procedure pedido_escribir( )
      { if (nr>0 OR nw>0)
                    { dw = dw +1;
                      wait (ok_escribir);
                    }
        else  nw = nw + 1;
      }

    procedure libera_escribir( )
     { if (dw > 0)
                { dw = dw -1;
                  signal (ok_escribir);
                }
       else { nw = nw -1;
              if (dr > 0)
                   { nr = dr;
                     dr = 0;
                     signal_all (ok_leer);
                   }
            }
      }
}
```

```
Process lector[id: 0..L-1]
{  while (true)
     { Controlador.pedido_leer();
       //Leer sobre la BD
       Controlador.libera_leer();
     };
}

Process escritor[id: 0..E-1]
{  while (true)
     { Controlador.pedido_escribir();
       //Leer sobre la BD
       Controlador.libera_escribir();
     };
}
```

Monitores con Pthreads

Ejemplo: *Lectores y escritores*

```
#include <pthread.h>

void *Escritor(void *);
void *Lector(void *);

int main(int argc, char * argv[ ])
{  int nr = 0, nw = 0, dr = 0, dw = 0, i;
   pthread_cond_t   ok_leer, ok_escribir;
   pthread_t  lectores[L], escritores[E];
   ……
   pthread_init();
   pthread_cond_init(&ok_leer, NULL);
   pthread_cond_init(&ok_escribir, NULL);
   ……
   for (i=0; i<E;i++) pthread_create (&escritores[i], &attr,  Escritor, NULL);
   for (i=0; i<L;i++) pthread_create (&lectores[i], &attr,  Lector, NULL);
}
```

Por cada monitor se require un *mutex* para similar la EM implicita de los mismos.

Monitores con Pthreads

Ejemplo: *Lectores y escritores*

- Como hay solo un monitor se pone solo una variable *mutex* que además de declarer se inicializa en el *main* del programa.

```
#include <pthread.h>

void *Escritor(void *);
void *Lector(void *);

int main(int argc, char * argv[ ])
{  int nr = 0, nw = 0, dr = 0, dw = 0, i;
   pthread_cond_t   ok_leer, ok_escribir;
   pthread_t  lectores[L], escritores[E];
   pthreads_mutex_t  mutex;
   ……
   pthread_init();
   pthread_cond_init(&ok_leer, NULL);
   pthread_cond_init(&ok_escribir, NULL);
   pthread_mutex_init(&mutex, NULL);
   ……
   for (i=0; i<E;i++) pthread_create (&escritores[i], &attr,  Escritor, NULL);
   for (i=0; i<L;i++) pthread_create (&lectores[i], &attr,  Lector, NULL);
}
```

Monitores con Pthreads

Ejemplo: *Lectores y escritores*

- Se agrega en los procesos el bloque y desbloqueo de *mutex* en los llamados a los procedure del monitor.

```
void *lector (void*)
{  while (true)
     { pthread_mutex_lock (&mutex);
       Controlador.pedido_leer();
       pthread_mutex_unlock (&mutex);
       //Leer sobre la BD
       pthread_mutex_lock (&mutex);
       Controlador.libera_leer();
       pthread_mutex_unlock (&mutex);
     }
}
```

```
void *escritor (void*)
{  while (true)
     { pthread_mutex_lock (&mutex);
       Controlador.pedido_escribir();
       pthread_mutex_unlock (&mutex);
       //Leer sobre la BD
       pthread_mutex_lock (&mutex);
       Controlador.libera_escribir ();
       pthread_mutex_unlock (&mutex);
     }
}
```

El próximo paso es reemplazar los llamados de los procedimientos por el código de los mismos

Monitores con Pthreads

Ejemplo: *Lectores y escritores*

```
void *escritor (void*)
{  while (true)
     { pthread_mutex_lock (&mutex);
       if (nr>0 OR nw>0)
         { dw = dw +1;
           pthread_cond_wait (& ok_escribir, &mutex);
         }
       else  nw = nw + 1;
       pthread_mutex_unlock (&mutex);
       //Escribe sobre la BD
       pthread_mutex_lock (&mutex);
       if (dw > 0)
         { dw = dw -1;
           pthread_cond_signal(&ok_escribir);
         }
       else
         { nw = nw -1;
           if (dr > 0)
             { nr = dr;
               dr = 0;
               pthread_cond_broadcast(&ok_leer);
             };
         };
       pthread_mutex_unlock (&mutex);
     };
   pthreads_exit();
};
```

```
void *lector(void*)
{  while (true)
     { pthread_mutex_lock (&mutex);
       if (nw>0)
         { dr = dr +1;
           pthread_cond_wait (& ok_leer, &mutex);
         }
       else  nr = nr + 1;
       pthread_mutex_unlock (&mutex);
       //Leer sobre la BD
       pthread_mutex_lock (&mutex);
       nr = nr – 1;
       if (nr == 0 and dw > 0)
         { dw = dw -1;
           pthread_cond_signal(&ok_escribir);
           nw = nw + 1;
         };
       pthread_mutex_unlock (&mutex);
     };
   pthreads_exit();
};
```

Librerías para manejo de PM

Operaciones Send y Receive

- Los prototipos de las operaciones son:

  Send (void *sendbuf, int nelems, int dest)

  Receive (void *recvbuf, int nelems, int source)

- Ejemplo:

P0

```
a = 100;
send(&a, 1, 1);
a = 0;
```

P1

```
receive(&a, 1, 0)
printf("%d\n", a);
```

- La semántica del SEND requiere que en P1 quede el valor 100 (no 0).

- Diferentes protocolos para Send y Receive.

Send y Receive bloqueante

- Para asegurar la semántica del SEND → no devolver el control del Send hasta que el dato a trasmitir esté seguro (Send bloqueante).
- Ociosidad del proceso.
- Hay dos posibilidades:
  - Send/Receive bloqueantes sin buffering.
  - Send/Receive bloqueantes con buffering.

Send y Receive no bloqueante

- Para evitar overhead (ociosidad o manejo de buffer) se devuelve el control de la operación inmediatamente.
- Requiere un posterior chequeo para asegurarse la finalización de la comunicación.
- Deja en manos del programador asegurar la semántica del SEND.
- Hay dos posibilidades:
  - Send/Receive no bloqueantes sin buffering.
  - Send/Receive no bloqueantes con buffering.

Message Passing Interface (MPI)

Librería MPI (Interfaz de Pasaje de Mensajes)

- Existen numerosas librerías para pasaje de mensaje (no compatibles).
- MPI define una librería estándar que puede ser empleada desde C o Fortran (y potencialmente desde otros lenguajes).
- El estándar MPI define la sintaxis y la semántica de más de 125 rutinas.
- Hay implementaciones de MPI de la mayoría de los proveedores de hardware.
- Modelo SPMD.
- Todas las rutinas, tipos de datos y constantes en MPI tienen el prefijo "MPI_". El código de retorno para operaciones terminadas exitosamente es MPI_SUCCESS.
- Básicamente con 6 rutinas podemos escribir programas paralelos basados en pasaje de mensajes: MPI_Init, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank, MPI_Send y MPI_Recv.

Librería MPI - Inicio y finalización de MPI

- ***MPI_Init***: se invoca en todos los procesos antes que cualquier otro llamado a rutinas MPI. Sirve para inicializar el entorno MPI.

```
MPI_Init (int *argc, char **argv)
```

  Algunas implementaciones de MPI requieren argc y argv para inicializar el entorno

- ***MPI_Finalize***: se invoca en todos los procesos como último llamado a rutinas MPI. Sirve para cerrar el entorno MPI.

```
MPI_Finalize ()
```

Librería MPI - Comunicadores

- Un comunicador define el dominio de comunicación.
- Cada proceso puede pertenecer a muchos comunicadores.
- Existe un comunicador que incluye a todos los procesos de la aplicación MPI_COMM_WORLD.
- Son variables del tipo MPI_Comm → almacena información sobre que procesos pertenecen a él.
- En cada operación de transferencia se debe indicar el comunicador sobre el que se va a realizar.

Librería MPI - Adquisición de Información

- ***MPI_Comm_size***: indica la cantidad de procesos en el comunicador.

  MPI_Comm_size (MPI_Comm comunicador, int *cantidad).

- ***MPI_Comm_rank***: indica el "rank" (identificador) del proceso dentro de ese comunicador.

  MPI_Comm_rank (MPI_Comm comunicador, int *rank)

  - rank es un valor entre [0..cantidad]
  - Cada proceso puede tener un rank diferente en cada comunicador.

EJEMPLO:

```
#include <mpi.h>

main(int argc, char *argv[])
 {  int cantidad, identificador;

    MPI_Init(&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &cantidad);
    MPI_Comm_rank(MPI_COMM_WORLD, &identificador);
    printf("Soy %d de %d \n", identificador, cantidad);
    MPI_Finalize();
 }
```

Librería MPI - Tipos de Datos para las comunicaciones

Tipo de Datos MPI	Tipo de Datos C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Librería MPI - Comunicación punto a punto

- Diferentes protocolos para Send.
  - Send bloqueantes con buffering (Bsend).
  - Send bloqueantes sin buffering (Ssend).
  - Send no bloqueantes (Isend).
- Diferentes protocolos para Recv.
  - Recv bloqueantes (Recv).
  - Recv no bloqueantes (Irecv).

Librería MPI - Comunicación bloqueante punto a punto

- MPI_Send, MPI_Ssend, MPI_Bsend: rutina básica para enviar datos a otro proceso.

MPI_Send (void *buf, int cantidad, MPI_Datatype tipoDato, int destino, int tag, MPI_Comm comunicador)

  - Valor de Tag entre [0..MPI_TAG_UB].

- MPI_Recv: rutina básica para recibir datos a otro proceso.

MPI_Recv (void *buf, int cantidad, MPI_Datatype tipoDato, int origen, int tag, MPI_Comm comunicador, MPI_Status *estado)

  - Comodines MPI_ANY_SOURCE y MPI_ANY_TAG._
  - Estructura MPI_Status

```
typedef struct MPI_Status { int MPI_SOURCE;
                            int MPI_TAG;
                            int MPI_ERROR; }
```

  - MPI_Get_count para obtener la cantidad de elementos recibidos

```
MPI_Get_count(MPI_Status *estado, MPI_Datatype  tipoDato, int *cantidad)
```

Ejemplo

Dos procesos intercambian valores (14 y 25). Solución empleando MPI:

```
# include  <mpi.h>
main (INT argc, CHAR *argv [ ] )  {
      INT id, idAux;
      INT longitud=1;
      INT valor, otroValor;
      MPI_status  estado;

      MPI_Init (&argc, &argv);

      MPI_Comm_Rank (MPI_COMM_WORLD, &id);
      IF (id == 0)  { idAux = 1; valor = 14;}
      ELSE { idAux = 0; valor = 25; }

      MPI_send (&valor, longitud, MPI_INT, idAux, 1, MPI_COMM_WORLD);
      MPI_recv (&otroValor, 1, MPI_INT, idAux, 1, MPI_COMM_WORLD, &estado);
      printf ("process %d received a %d\n", id, otroValor);
      MPI_Finalize ( );
 }
```

Ejemplo

En este caso resolvemos el mismo ejercicio pero para que no haya Deadlock si el Send actúa como Ssend.

```
# include  <mpi.h>
main (INT argc, CHAR *argv [ ] )  {
      INT id;
      INT valor, otroValor;
      MPI_status  estado;

      MPI_Init (&argc, &argv);
      MPI_Comm_Rank (MPI_COMM_WORLD, &id);
      IF (id == 0)  { valor = 14;
                        MPI_send (&valor, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
                        MPI_recv (&otroValor, 1, MPI_INT, 1, 1, MPI_OMM_WORLD, &estado);
                      }
      ELSE { valor = 25;
               MPI_recv (&otroValor, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &estado);
               MPI_send (&valor, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
             }
      printf ("process %d received a %d\n", id, otroValor);
      MPI_Finalize ( );
 }
```

Librería MPI - Comunicación no bloqueante punto a punto

- Comienzan la operación de comunicación e inmediatamente devuelven el control (no se asegura que la comunicación finalice correctamente).

```
MPI_Isend (void *buf, int cantidad, MPI_Datatype tipoDato, int destino, int tag, MPI_Comm comunicador,
        MPI_Request *solicitud)
```

```
MPI_Irecv (void *buf, int cantidad, MPI_Datatype tipoDato, int origen, int tag, MPI_Comm comunicador,
        MPI_Request *solicitud)
```

- MPI_Test: testea si la operación de comunicación finalizó.

```
MPI_Test (MPI_Request *solicitud, int *flag, MPI_Status *estado)
```

- MPI_Wait: bloquea al proceso hasta que finaliza la operación.

```
MPI_Wait (MPI_Request *solicitud, MPI_Status *estado)
```

- Este tipo de comunicación permite solapar computo con comunicación. Evita overhead de manejo de buffer. Deja en manos del programador asegurar que se realice la comunicación correctamente.

Librería MPI - Comunicación no bloqueante punto a punto

Código usando comunicación bloqueante

```
EJEMPLO: main (int argc, char *argv[])
          {  int cant, id, *dato, i;
             MPI_Status estado;

             dato = (int *) malloc (100 * sizeof(int));
             MPI_Init(&argc,&argv);
             MPI_Comm_rank (MPI_COMM_WORLD, &id);

             if (id == 0)
               { cant = atoi(argv[1])%100;
                 MPI_Send(dato,cant,MPI_INT,1,1,MPI_COMM_WORLD);
                 for (i=0; i< 100; i++) dato[i]=0;
               }
             else
               { MPI_Recv(dato,100,MPI_INT,0,1,MPI_COMM_WORLD, &estado);
                 MPI_Get_count(&estado, MPI_INT, &cant);
                 //PROCESA LOS DATOS;
               };
             MPI_Finalize;
          }
```

Para usar comunicación NO bloqueante (¿alcanza con cambiar el Send por Isend?)

Librería MPI - Comunicación no bloqueante punto a punto

Código anterior usando comunicación no bloqueante

```
EJEMPLO: main (int argc, char *argv[])
          {  int cant, id, *dato, i;
             MPI_Status estado;
             MPI_Request req;

             dato = (int *) malloc (100 * sizeof(int));
             MPI_Init(&argc,&argv);
             MPI_Comm_rank (MPI_COMM_WORLD, &id);

             if (id == 0)
               { cant = atoi(argv[1]);
                 //INICIALIZA dato
                 MPI_Isend(dato,cant,MPI_INT,1,1,MPI_COMM_WORLD, &req);
                 //TRABAJA
                 MPI_Wait(&req, &estado);
                 for (i=0; i< 100; i++) dato[i]=0;
               }
             else
               { MPI_Recv(dato,100,MPI_INT,0,1,MPI_COMM_WORLD, &estado);
                 MPI_Get_count(&estado, MPI_INT, &cant);
                 //PROCESA LOS DATOS;
               };
             MPI_Finalize;
          }
```

Librería MPI - Comunicación no bloqueante punto a punto

```
EJEMPLO: main (int argc, char *argv[])
          {  int id, *dato, i, flag;
             MPI_Status estado;
             MPI_Request req;

             dato = (int *) malloc (100 * sizeof(int));
             MPI_Init(&argc,&argv);
             MPI_Comm_rank (MPI_COMM_WORLD, &id);

             if (id == 0)
               { //INICIALIZA dato
                 MPI_Send(dato,cant,MPI_INT,1,1,MPI_COMM_WORLD);
               }
             else
               { MPI_Irecv(dato,100,MPI_INT,0,1,MPI_COMM_WORLD ,&req);
                 MPI_Test(&req, &flag,&estado);
                 while (!flag)
                   { //Trabaja mientras espera
                     MPI_Test(&req, &flag,&estado);
                   };
                 //PROCESA LOS DATOS;
               };
             MPI_Finalize;
          }
```

Librería MPI – Consulta de mensajes pendientes

- Información de un mensaje antes de hacer el Recv (Origen, Cantidad de elementos, Tag).
- MPI_Probe: bloquea el proceso hasta que llegue un mensaje que cumpla con el origen y el tag.

```
MPI_Probe (int origen, int tag, MPI_Comm comunicador, MPI_Status *estado)
```

- MPI_Iprobe: cheqea por el arribo de un mensaje que cumpla con el origen y tag.

```
MPI_Iprobe (int origen, int tag, MPI_Comm comunicador, int *flag, MPI_Status *estado)
```

- Comodines en Origen y Tag.

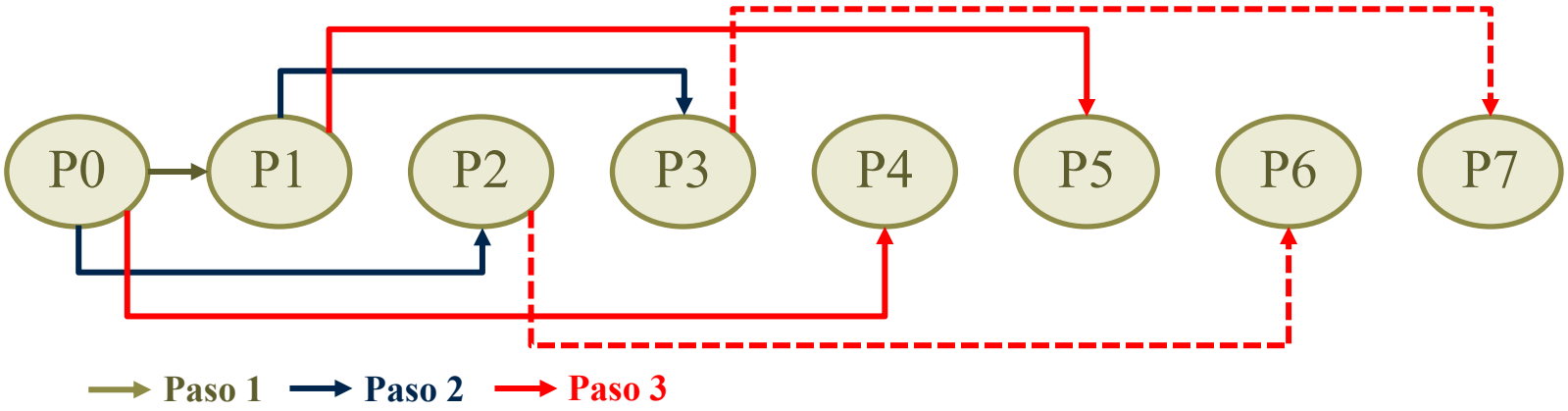
¿Cuando y porque usar cada uno?

Librería MPI - Comunicaciones Colectivas

MPI provee un conjunto de funciones para realizar operaciones colectivas, sobre un grupo de procesos asociado con un comunicador. Todos los procesos del comunicador deben llamar a la rutina colectiva:

- MPI_Barrier
- MPI_Bcast
- MPI_Scatter - MPI_Scatterv
- MPI_Gather - MPI_Gatherv
- MPI_Reduce
- Otras…

Ventajas del uso de comunicaciones colectivas.



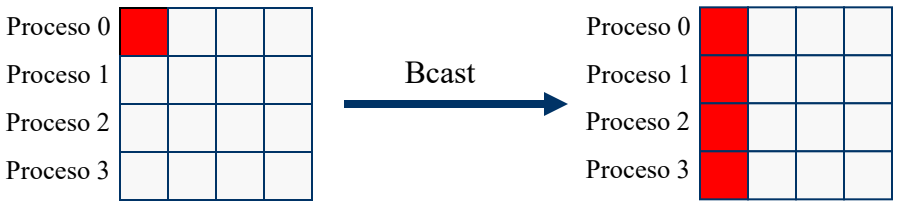
Librería MPI - Comunicaciones Colectivas

- Sincronización en una barrera.

MPI_Barrier(MPI_Comm comunicador)

- Broadcast: un proceso envía el mismo mensaje a todos los otros procesos (incluso a él) del comunicador.

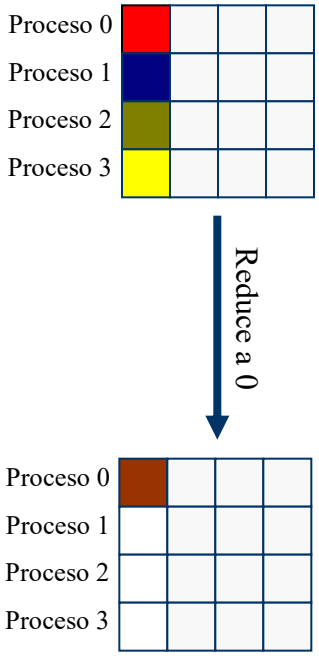
MPI_Bcast (void *buf, int cantidad, MPI_Datatype tipoDato, int origen, MPI_Comm comunicador)



Librería MPI - Comunicaciones Colectivas (cont.)

- Reducción de todos a uno: combina los elementos enviados por cada uno de los procesos (inclusive el destino) aplicando una cierta operación.

MPI_Reduce (void *sendbuf, void *recvbuf, int cantidad, MPI_Datatype tipoDato, MPI_Op operación, int destino , MPI_Comm comunicador)



Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

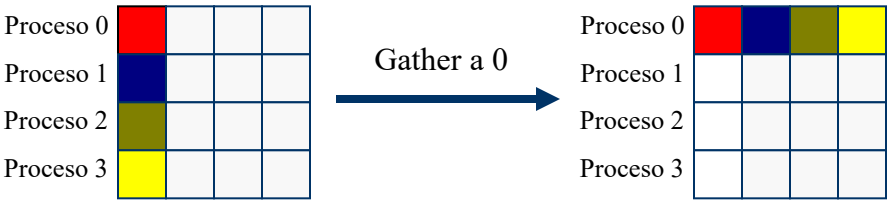
Librería MPI - Comunicaciones Colectivas (cont.)

- Gather: recolecta el vector de datos de todos los procesos (inclusive el destino) y los concatena en orden para dejar el resultado en un único proceso.
  - Todos los vectores tienen igual tamaño.

MPI_Gather (void *sendbuf, int cantEnvio, MPI_Datatype tipoDatoEnvio, void*recvbuf, int cantRec, MPI_Datatype tipoDatoRec, int destino, MPI_Comm comunicador)

  - Los vectores pueden tener diferente tamaño.

MPI_Gatherv (void *sendbuf, int cantEnvio, MPI_Datatype tipoDatoEnvio, void*recvbuf, int *cantsRec, int *desplazamientos, MPI_Datatype tipoDatoRec, int destino, MPI_Comm comunicador)



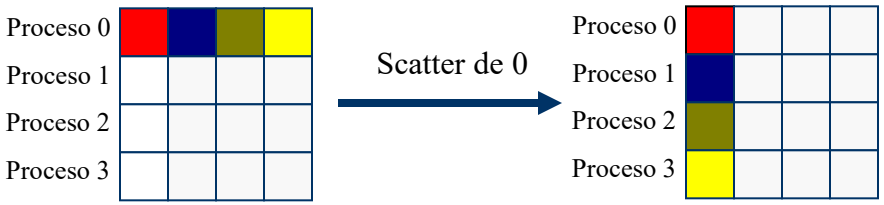
Librería MPI - Comunicaciones Colectivas (cont.)

- Scatter: reparte un vector de datos entre todos los procesos (inclusive el mismo dueño del vector).
  - Reparte en forma equitativa (a todos la misma cantidad).

```
MPI_Scatter (void *sendbuf, int cantEnvio, MPI_Datatype tipoDatoEnvio, void*recvbuf, int cantRec,
             MPI_Datatype tipoDatoRec, int origen, MPI_Comm comunicador)
```

  - Puede darle a cada proceso diferente cantidad de elementos.

```
MPI_Scatterv (void *sendbuf, int *cantsEnvio, int *desplazamientos, MPI_Datatype tipoDatoEnvio,
              void*recvbuf, int cantRec, MPI_Datatype tipoDatoRec, int origen,
              MPI_Comm comunicador)
```



Minimizando los overheads de comunicación.

- Maximizar la localidad de datos.
- Minimizar el volumen de intercambio de datos.
- Minimizar la cantidad de comunicaciones.
- Considerar el costo de cada bloque de datos intercambiado.
- Replicar datos cuando sea conveniente.
- Lograr el overlapping de cómputo (procesamiento) y comunicaciones.
- En lo posible usar comunicaciones asincrónicas.
- Usar comunicaciones colectivas en lugar de punto a punto